Task 1: Repository Layer – Appointment Methods

Objective

Create methods that handle fetching appointment-related data from the database.

Requirements

- Create Appointment Repository
- Implement the following methods in the repository class:
  - `retrieveDepartments()` – fetch all departments from the database
  - `retrieveProceduresByDepartment(departmentId : Int)` – fetch procedures for a selected department
  - `retrieveDoctorsByDepartment(departmentId : Int)` – fetch doctors and their ratings for the selected department
  - `retrieveAllAppointmentsByDoctorAndDate(doctorId : Int, date: DateTime)`

Technical

- Follow MVC architecture and OOP principles strictly
- Methods should interact directly with the database

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 2: Repository Layer – Time Slots Logic

Objective

Create logic to retrieve available time slots based on selected operation duration and chosen date.

Requirements

- `retrieveAvailableTimeSlots(procedureId : Int, date : DateTime)` – Fetch available 30-minute time slots, filtering by the duration of the procedure (e.g. 1-hour procedures should return only two consecutive slots).

Technical

- Ensure database queries handle filtering logic
- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 3: Repository Layer – Appointment Registration

Objective

Create logic to register a new appointment in the database.

Requirements

- Implement method:
  - `addAppointment(appointment: Appointment)` – Insert the new appointment into the database
- Ensure method handles exceptions (database issues, validation errors)

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Make sure the list of Consultations is an ObservableCollection or smth. similar that tracks the updates in the Repository and database
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 4: Service Layer – Department & Procedure Methods

Objective

Create service-layer methods to provide data for UI from the repository.

Requirements

- Create methods:
  - `getDepartment()` – Calls repository and returns department data
  - `getProceduresByDepartment(departmentId : Int)` – Wrap repository method to fetch operations

Technical

- Validate the date on the client side to ensure it is within the allowed 30-day window
- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Make sure the repository methods communicate correctly with the database without altering data.
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 5: Service Layer – Doctors & Time Slots Methods

Objective

Handle business logic for retrieving doctors and available time slots.

Requirements

- Create methods:
  - `getDoctorsByDepartment(departmentId : Int)` – Fetch and prepare doctor data (including rating)
  - `getAvailableTimeSlots(procedureId : Int, date : DateTime)` – Wrap repository method for fetching and potentially further filtering slots

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Maintaing clear error handling

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 6: Service Layer – Appointment Creation Logic

Objective

Implement validation and logic for registering new appointments.

Requirements

- Create method:
  - `createAppointment(appointment : Appointment)` – Validates data and calls `addAppointment()` from repository
- Method must validate that all required fields (department, procedure, doctor, date, slot) are provided and validated

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Return clear, useful responses or exceptions to the UI layer

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 7: UI Layer (WinUI) – Department and Procedure Selection Screen

Objective

Develop the initial selection UI (departments and operations).

Requirements

- Create a WinUI Page:
    - Display departments in a list view
    - On department selection, populate another list view with procedures

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Ensure responsive layout following the app design

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 8: UI Layer (WinUI) – Doctor selection with ratings display

Objective

Develop the UI component for displaying doctors and their ratings.

Requirements

- On operation selection, display a list of doctors and their ratings
- Use a clear and intuitive design (rating stars or numeric rating)

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Ensure responsive layout following the app design

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 9: UI Layer (WinUI) – Appointment Date & Time slots selection

Objective

Implement UI components for selecting valid dates and time slots.

Requirements

- Integrate a Calendar control to select dates (limited to tomorrow + 30 days)
- Dynamically display available time slots based on selected procedure duration

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Ensure responsive layout following the app design
- Validate user-selected dates client-side.

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 10: UI Layer (WinUI) – Appointment Confirmation Button

Objective

Enable a UI element for users to confirm appointments after all required fields are selected.

Requirements

- Implement a confirmation button enabled only after department, procedure, doctor, date and time slot are selected and validated

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Ensure responsive layout following the app design
- Validate user-selected dates client-side.
- Provide visual feedback (disabled/enabled button clearly visible).

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 11: UI Layer (WinUI) – Appointment Confirmation Modal

Objective

Create a modal or screen for confirming final appointment details.

Requirements

- Display appointment summary (department, procedure, doctor, date, time slot)
- Implement confirmation and cancellation options clearly

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Ensure responsive layout following the app design
- Validate user-selected dates client-side.
- Provide visual feedback (disabled/enabled button clearly visible).

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 12: UI Layer (WinUI) – Confirmation Message UI

Objective

Implement a user-friendly confirmation screen/message after successful appointment registration.

Requirements

- Show a confirmation message with a success message and key appointment details

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Ensure responsive layout following the app design
- Validate user-selected dates client-side.
- Implement simple, clear UI elements.

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 13: Integration & Testing

Objective

Integrate all layers and ensure complete and smooth functionality.

Requirements

- Connect all UI elements to respective service methods
- Test the entire flow (Department -> Procedure -> Doctor -> Date -> Time Slot -> Confirm -> Success Message)
- Implement and handle edge cases and exceptions (e.g. duplicate appointment attempts, invalid input)

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code
- Log errors and exceptions clearly.

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 1: Implement Fetching Upcoming Appointments in the Service Layer

Objective

Ensure the service layer retrieves all upcoming appointments for the currently logged-in patient.

Requirements

- Implement `getAppointmentsForPatient(patientID: int)` in AppointmentService.
- Use the existing method from the repository (`getAllAppointmentsByPatient(patientID: int)`).
- Ensure the method only returns future appointments, filtering out past appointments.
- Return a structured list of appointment objects, ready for processing in the UI.
- Raise an exception if no upcoming appointments exist.

Technical

- Follow MVC principles (Model → Repository → Service → UI).
- Ensure error handling for database failures.
- Ensure asynchronous execution for smooth UI interactions.

Final

- The service layer correctly fetches upcoming appointments, ready to be used in the calendar view.

Task 2: Display Appointments in the Calendar UI

Objective

Ensure the UI correctly displays appointments in a calendar format.

Requirements

- Integrate `CalendarView` in the UI to display appointments.

- Use `AppointmentService.getAppointmentsForPatient(patientID: int)` to fetch appointments.
- Highlight dates containing at least one appointment.
- Make sure each appointment is correctly mapped to its date.
- Ensure UI dynamically updates when new appointments are loaded.

Technical

- The UI should only communicate with the Service Layer, not the repository.
- Implement an observable list to track dynamic updates.
- Ensure responsiveness across different screen sizes.

Final

- The calendar correctly highlights dates with appointments, allowing users to see their upcoming visits at a glance.

Task 3: Implement Clickable Appointments in the Calendar UI

Objective

Allow users to click a date in the calendar to see all appointments for that day.

Requirements

- When a date is clicked, list all appointments scheduled for that day.
- Display key details (time, doctor, type, location).
- Ensure that appointment details are dynamically updated if a new appointment is added.
- Implement an event listener to detect when a date is selected.

Technical

- Use `CalendarEvent` to detect date selection.
- Retrieve appointments via `AppointmentService.getAppointmentsForPatient(patientID: int)`.
- Implement data binding to ensure updates reflect immediately in the UI.

Final

- Clicking a date correctly displays the list of appointments for that day, enhancing user experience.

Task 4: Implement Appointment Details Overlay

Objective

When an appointment is clicked, show an overlay/modal displaying full appointment details.

Requirements

- Implement a modal overlay that displays:
  - Date & Time
  - Doctor's Name & Department
  - Operation Type
  - Location
- Call `method to get data` from the service layer/data base.
- Include a close button to return to the calendar view.

Technical

- Follow the MVC structure (Service should be the only layer interacting with the repository).
- Use `ObservableCollection` to ensure real-time updates.
- Implement data validation to avoid errors.

Final

- Patients can click on an appointment to view its details, improving usability.

Task 5: Implement Appointment Cancellation Button with 24-Hour Rule

Objective

Ensure that the appointment details modal displays a Cancel Appointment button that follows the 24-hour cancellation rule.

Requirements

- Display a Cancel Appointment button inside the appointment details modal.
- The button should always call cancelAppointment(appointmentID: int) from the service layer.
- If the appointment is less than 24 hours away, disable the button (greyed-out and unclickable).
- If the button is disabled, it should still be visible, but unclickable.
- Display confirmation messages for successful cancellations.
- Show an error message if cancellation is not allowed.

Technical

- Implement event handling for cancellation.
- Use WinUI ContentDialog for the modal window.
- The UI should only interact with the Service Layer, not the Repository directly.
- Dynamically update the button state based on appointment time.
- Ensure proper error handling from the service layer response.

Final

- The modal correctly displays appointment details.
- The Cancel Appointment button dynamically updates its state based on the 24-hour rule.
- Calls to the service layer are correctly handled.
- The UI remains responsive and functional.

Task 6: Handle Edge Cases for Appointment Retrieval and Cancellation

Objective

Ensure robust error handling and improve data integrity for appointment retrieval and cancellation.

Requirements

- If `getAppointmentsForPatient(patientID: int)` returns an empty list, display a "No upcoming appointments" message.
- Handle database connection failures gracefully with fallback error messages.
- Ensure appointment cancellation logs errors for debugging if the database operation fails.

Technical

- Implement try-catch blocks to prevent UI crashes.
- Ensure proper validation before attempting cancellation.
- Log errors for debugging purposes.

Final

- The system handles error scenarios gracefully, ensuring a smooth user experience.

Task 7: Implement Automatic Calendar Refresh Using Observer Pattern

Objective

Ensure that the calendar view automatically updates every 5 seconds to display any new appointments.

Requirements

- Implement an observer mechanism that refreshes the calendar UI every 5 seconds.
- Fetch all upcoming appointments from `AppointmentService.getAppointmentsForPatient(patientID: int).`
- Do not check if the data has changed—always refresh everything.
- Ensure that the refresh does not interfere with user interactions (e.g., scrolling or selecting an appointment).

Technical

- Use ObservableCollection for dynamic updates.
- Implement a background timer (Task.Delay or DispatcherTimer in C#) to fetch data every 5 seconds.
- Ensure asynchronous execution to avoid UI freezing.
- The UI should only interact with the Service Layer, not the Repository directly.

Final

- The calendar view automatically updates every 5 seconds, ensuring that patients always see the latest appointment data.

Task 8: Test and Validate the "See My Upcoming Appointments" Feature

Objective

Conduct testing to validate the functionality of the appointments feature in different scenarios.

Requirements

- Test fetching upcoming appointments under the following conditions:
  - Appointments exist → Calendar should highlight dates correctly.
  - No appointments exist → UI should display "No upcoming appointments."
  - Database failure → UI should display an error message.
- Test appointment selection:
  - Clicking a date should correctly list all appointments for that day.
- Test appointment cancellation:
  - Cancellation should succeed only if the appointment is more than 24 hours away.
  - An error should be displayed if an appointment is too close to the due time.

Technical

- Perform manual UI testing to verify correct display.
- Implement unit tests for `getAppointmentsForPatient()` and `cancelAppointment()`.

- Test database interactions to ensure correct data retrieval and modification.

Final

- The entire feature is tested and validated, ensuring correctness and reliability.

This task ensures that the calendar remains up-to-date in real-time without requiring manual refreshes. Let me know if you want me to compile everything into a Word document for structured task management!

Task 3.a: Medical Record Repository

Objective

Create a repository that contains the necessary functions that communicate with the database and provide structured results for upper layers.

Requirements

- Create a Repository for the “MedicalRecord” Entity
- Connect the repository to the local SQLServer
- Implement all CRUD operations(add, delete, update)
- Create the “retrieveAllConsultations(int PatientId)” method which will return the list of all Consultations of a patient
- Create the “retrieveConsultationById(int MedicalRecordId)” method inside it, which takes as input the id of a consultation and returns the consultation having that ID

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 3.b: Interface & Service

Objective

Create a service class, which will handle the business logic of the app and will use the repository to get data and provide it to the UI in the way it is intended to.

Requirements

- Create the Service for the “MedicalRecord”
- Create the “getAllConsultations” method that will use the appropriate method from the repository to pass the Consultations to the UI
- Create the “getConsultationById” method that receives a MedicalRecord Id and uses the appropriate method from the repository to pass the requested Consultation to the UI
- Inter-layer communication should be performed using only interfaces from lower layers. These interfaces will contain all the methods necessary for upper layers to work properly. At the moment, the UI layer should use a Service interface (e.g. IMedicalRecordService) and the Service layer should use a Repository interface (e.g. IMedicalRecordRepository). Each repository of each entity will implement the repository interface corresponding to that entity.

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 3.c: Create a WinUI Window for Viewing Consultation Objects

Objective

Develop a new WinUI window that displays a list of Consultation objects.

Requirements

- Create a new WinUI window called History
- Add a ListView or something similar to achieve a list-form display of objects
- The list of Consultations should be taken from the repository and should contain all the logged in patient's consultations
- Create a card component which will be the basic list item
- For each consultation display on the card the following "Consultation – {Date}" and beneath this text the department of the doctor associated

Technical

- Respect OOP principles and use MVVC and MVC architecture (model-repo-service-ui)
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 3.d: Sync the displayed list with db info

Objective

Research & implement the MVVC pattern in C# and create the HistoryViewModel class

Requirements

- Research Observable classes in C#
- Implement the HistoryViewModel class that contains an ObservableList<MedicalRecord> property that will be bound to the View
- Implement the LoadMedicalRecords(int PatientId) function which retrieves the list of all MedicalRecords of a patient and stores it in the ObservableList property of the class
- Bind this model to the view to display the observable list property

Technical

- Respect OOP principles and use MVVC architecture
- Specify & test your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 3.e: Make the Medical Record Details Modal

Objective

Now that the list is functional, we need to make the individual cards clickable. Clicking one card will open a modal window which will show all attributes/details (minus doctor and user Ids) of the clicked consultation.

Requirements

- Create an onClick function for the card component of the list which opens the new modal component
- The modal should contain the consultation date, doctor name, doctor department, operation name, and a list of strings representing the resulted documents (ex. Res1.pdf, Res2.pdf)
- The way these are displayed and the design of the modal are left to your choice as long as they fit in with the rest of the app
- Add any new service/repository functions to get the needed data if the current ones are not enough
- Add a button to be able to download the result documents from the database.

Technical

- Respect OOP principles and use MVVC architecture
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 3.f: Make the download button work

Objective

The aim is to imitate the transfer of documents as presented in the Assignment2.

Requirements

- The database will only store the local path of the document on the computer for now
- Create an onClick function for the download button in the Consultation Details Modal
- Implement the download functionality by getting the document paths from the database and copy these documents in the Downloads folder of the PC
- Implement all necessary functions in the service and repo to get the documents associated with the consultation id from the db if they are not already implemented
- Look at the UML for details

Technical

- Respect OOP principles and use MVC architecture (model-repo-service-ui)
- Specify your code

Final

- The requirements are flexible as long as you achieve the desired functionalities and the project leader agrees with your approach

Task 1.a: Repository

Objective

Create a repository that contains the necessary functions that communicate with the database and provide structured results for upper layers.

Requirements

- Create Repository for the “Shift” entity
- Implement the “retrieveShiftByDoctorAndDate(doctorID:int, date:date)” method, which:
    - takes a doctor’s ID and a date as input
    - returns a Shift object for that day
    - throw an error if no shift exists.
- Implement all CRUD operations for the ShiftRepository.
- Implement the “retrieveAllAppointmentsByDoctorAndDate (doctorID:int, date:date)”, which:
    - returns a list containing all the Appointment objects that are suitable
    - returns an empty list if no Appointment exist

Technical

- Respect OOP principles and Follow MVC Architecture (Model → Repository → Service → UI)
- Ensure code is well-documented and error-handling is in place

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.b: Service

Objective

Create a service class, which will handle the business logic of the application and will use the repository to get data and provide it to the UI in the way it is intended to.

Requirements

- Create the Service for the “Shifts”
- Create the “retrieveShiftForDoctorAndDate(doctorID:int , date:date)” method which will:
    - take a doctor’s ID and a date as input
    - uses the appropriate method from the repository to get the Shift corresponding to the doctorID and the date
- Create the “retrieveAllAppointmentsByDoctorAndDate (doctorID:int, date:date)” method which will:
    - take a doctor’s ID and a date as input
    - uses the appropriate method from the repository to get the list of Appointment objects corresponding to the doctorID and the date
- Inter-layer communication should be performed using only interfaces from lower layers. These interfaces will contain all the methods necessary for upper layers to work properly. Now, the UI layer should use a Service interface (e.g IShiftService) and the Service layer should use a Repository Interface (e.g IShiftRepository). Each Repository of each entity will implement the repository corresponding to that entity.
- Add any new service/repository functions needed to get the needed data if the current ones are not enough

Technical

- Respect OOP principles and Follow MVC Architecture (Model → Repository → Service → UI)
- Ensure code is well-documented and error-handling is in place

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.c: Setting Up the CalendarView and UI Elements

Objective

Develop a WinUI window that displays a monthly calendar where the doctor can view their work shifts and scheduled appointments.

Requirements

- Create a new WinUI window called ScheduleCalendar
- Add a CalendarView for the monthly display
- Apply appropriate styling to the calendar for visual clarity and usability.

Technical

- Follow MVVM for UI and MVC Architecture for backend (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.d: Implementing Selection & Navigation Time Slots

Objective

Implement date selection on the calendar and navigation to display the selected day's appointments and shifts.

Requirements

- Implement a selection event (date selection handler) for the CalendarView.
- On selecting a date, navigate to the DailySchedule window for that day.
- Pass the selected date as a parameter to the DailySchedule view.
- Ensure the navigation transition is smooth and validate the date parameter which is passed.

Technical

- Follow MVVM for UI and MVC Architecture for backend (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.e: Sync the calendar data with the database information

Objective

Research and implement the MVVC pattern in C# and create the DoctorCalendarViewModel class

Requirements

- Research ObservableCollection<T> in C# for real-time UI updates.
- Implement the DoctorCalendarViewModel which will contain an
    - ObservableCollection<Appointment>
    - ObservableCollection<Shift>
    - Be bound to the View for automatic updates
- Implement the LoadDoctorCalendarDataAsync(int doctorId) function to:
    - fetch the doctor's shifts and appointments from the database
    - store the data in the ObservableCollections
- Ensure DoctorCalendarViewModel implements "notify()" function where necessary.

Technical

- Follow MVVM for UI and MVC Architecture for backend (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.f: Creating the DailySchedule Window & Time Slot Structure

Objective

Develop a WinUI window that displays a list of 30-minute time slots for a selected day.

Requirements

- Create a new WinUI window called DailySchedule
    - The window should open after selecting a date from the calendar
    - It should display a list of 30-minutes time slots for that day
- Each slot should represent 30 minutes, covering 24 hours (48 slots total)
- Time slots should be empty placeholders initially (data will be loaded in the next task)
- Use ListView / GridView for rendering time slots.

Technical

- Follow MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.g: Retrieving & Displaying Appointments & Work Shifts

Objective

Modify the DailySchedule window to fetch and display work shifts and appointments dynamically

Requirements

- Retrieve Work Shifts and Appointments using the Service Layer
    - retrieveAllAppointmentsByDoctorAndDate(doctorId, date)
    - retrieveShiftForDoctorAndDate(doctorId, date)
- Populate Time Slots with Data
    - If a time slot within a work shift, colour it red
    - If a time slot has an appointment within a work shift, colour it yellow and show patient details
    - If a time slot is free, keep it at its default colour.
- Ensure data updates dynamically
    - Use an ObservableColllection so that changes to shifts or appointments update the UI automatically.

Technical

- Follow MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.h: Handling User Interactions & Click Events

Objective

Implement clickable behaviour for time slots, allowing doctors to interact with appointments and shifts.

Requirements

- Define Click Behaviour for different slot types.
- Clicking on an Appointment slot will show
    - A “Create Medical Record” button
    - A “View Profile” button
    - A “Medical Records History” button
    - A “Appointments details” button
- Clicking on a Work Shift Slot (without an appointment) should display:
    - “No appointments scheduled in this time slot.”
- Clicking on an Empty Slot should do nothing
- Ensure data updates dynamically

Technical

- Follow MVVM and MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.
- The repository methods should retrieve data without altering database records.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.i: Implementing the Appointment Actions UI

Objective

Create a new WinUI window that appears when clicking an appointment slot, displaying the four action buttons.

Requirements

- Create a new WinUI window called AppointmentActionsWindow.
- Display four buttons inside this window:
    - A “Create Medical Record” button -> Navigates to the Medical Record Form.
    - A “View Profile” button -> Navigates to the Patient’s Profile.
    - A “Medical Records History” button -> Opens the Patient’s Consultation History.
    - A “Appointments details” button -> Opens a detailed view of the appointment
- When an appointment slot is clicked, this window should appear
- Clicking each button should navigate to the respective section

Technical

- Follow MVVM and MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.
- The repository methods should retrieve data without altering database records.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.j: Implement “Create Medical Record” button

Objective

Allow doctors to create a medical record for a selected appointment

Requirements

- Add a “Create Medical Record” button to the UI.
    - Bind the button to a command in the ViewModel
- Implement a click event handler for the button.
- When clicked, the system should navigate to the medical record creation form -> Task2.Doctor
- Ensure the UI interacts only with the Service Layer to retrieve appointment details.
- Pass the selected appointment details to the new form (for autofill)

Technical

- Follow MVVM and MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.
- The repository methods should retrieve data without altering database records.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.k: Implement “Medical Records History” button

Objective

Allow doctors to view the patient’s past consultations.

Requirements

- Add a “Medical Record History” button to the UI.
- Implement a click event handler for the button.
- When clicked, the system should navigate to the medical record history. -> Task3.Patient
- Ensure the UI interacts only with the Service Layer to retrieve appointment details.

Technical

- Follow MVMV and MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.l: Implement “View Profile” button

Objective

Allow doctors to view the patient’s full profile.

Requirements

- Add a “View Profile” button.
- Implement a click event handler for the button.
- When clicked, the system should navigate to the user dashboard. -> Another group’s task
- Ensure the UI interacts only with the Service Layer to retrieve appointment details.

Technical

- Follow MVMV and MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Task 1.m: Implement “Appointment’s details” button

Objective

Allow doctors to view detailed information about a selected appointment.

Requirements

- Add a “View Profile” button.
- Implement a click event handler for the button.
- When clicked, the system should navigate to the appointment’s dashboard. -> Another group’s task
- Ensure the UI interacts only with the Service Layer to retrieve appointment details.
- The repository methods should retrieve data without altering existing database records.

Technical

- Follow MVMV and MVC Architecture (Model → Repository → Service → UI)
- The UI should only interact with the Service Layer, not the Repository directly
- Make sure the repository methods communicate correctly with the database without altering data.

Final

- The requirements are flexible if the desired functionalities are achieved, and the project leader agrees with the approach.

Doctor 2:

Task 2.a: Designing the MedicalRecord Form UI

Objective

Create a UI form where doctors can input MedicalRecord details.

Requirements

- Develop a **WinUI** form called **CreateMedicalRecordWindow**.
- Include input fields for:
    - **Procedure Name (drop down)**
    - **Conclusion** (optional, can be updated later)
- Auto-fill doctor's ID, patient's ID, and MedicalRecord's ID.
- Add a **"Submit"** button for saving the MedicalRecord's.
- Implement basic form validation.

Technical

- Follow MVC architecture.
- Use **data binding** for form fields (to pre-fill details).
- Ensure proper input validation before submission.
- Implement UI responsiveness for different screen sizes.

Final

- The form should be fully functional, visually clear, and properly pre-filled.
- Submission should be disabled until valid input is provided.

Task 2.b: Handling Document Uploads in Backend

Objective

Ensure uploaded documents are correctly stored and linked to MedicalRecords.

+Documentation about this

Requirements

- Implement a **file storage service**.
- Store files securely in a designated location.
- Link stored files to the MedicalRecord in the database.

Technical

- Use **cloud storage** or **local server storage**.
- Implement **file encryption** if necessary.

Final

- Doctors should be able to retrieve and review uploaded documents later.

Task 2.c: Implementing Document Upload in the MedicalRecord Form

Objective

Enable doctors to attach documents (test results, reports, etc.) to the MedicalRecord.

Requirements

- Add a **file upload component** in the MedicalRecord form.
- Allow multiple document uploads.
- Restrict file types to **PDF, JPG, PNG, DOCX**.
- Display uploaded documents in a list.
- Ensure documents are linked to the MedicalRecord in the database.

Technical

- Follow MVC architecture.
- Implement file handling logic in the **Service Layer**.
- Ensure secure file storage in the backend.
- Validate file size limits before upload.

Final

- Doctors should be able to upload, view, and remove documents before submitting.
- Files should be stored securely and linked to MedicalRecords.

Task 2.d: Implementing `addMedicalRecord(c: MedicalRecord)` in Repository

Objective

Develop a repository function to store MedicalRecord data in the database.

Requirements

- Implement `addMedicalRecord(c: MedicalRecord)` in the repository.
- Check if a MedicalRecord already exists for the appointment.
- Insert the MedicalRecord into the database.
- Throw an error in case of failure or duplicate MedicalRecord.

Technical

- Follow MVC architecture.
- Use **SQL transactions** to ensure atomic database procedures.
- Implement error handling and logging.

Final

- The function should correctly store MedicalRecord data and prevent duplicates.
- Database integrity should be maintained.

Task 2.e: Retrieving MedicalRecord by Appointment

Objective

Allow fetching an existing MedicalRecord for a given appointment.

Requirements

- Implement `retrieveMedicalRecordForAppointment(appointmentId)` in the repository.
- If a MedicalRecord exists, return its details.
- If no MedicalRecord exists, return `null`.

Technical

- Use **parameterized queries** to prevent SQL injection.
- Ensure efficient database querying for performance.

Final

- Doctors should be able to retrieve and view MedicalRecord details if they exist.

Task 2.f: Implementing `addMedicalRecord(c: MedicalRecord)` in Service Layer

Objective

Enable the Service Layer to add MedicalRecords using the repository.

Requirements

- Implement `addMedicalRecord(c: MedicalRecord)` in the **Service Layer**.
- Call the repository method to store the MedicalRecord.
- Handle validation and errors before saving.

Technical

- Follow MVC architecture.
- Implement **try-catch** blocks for error handling.
- Ensure **data consistency** across the application.

Final

- The service should correctly process and pass MedicalRecords to the repository.

Task 2.g: Handling MedicalRecord Submission from UI

Objective

Ensure MedicalRecord details are correctly saved when the doctor presses submit.

Requirements

- Validate form fields before submission.
- Call `addMedicalRecord(c: MedicalRecord)` in the Service Layer.
- Display success or error messages based on submission result.

Technical

- Implement **asynchronous form submission** to avoid UI freezing.
- Ensure database constraints are enforced on submission.

Final

- MedicalRecords should be successfully saved, and the UI should reflect the changes.

Task 2.h: Auto-Filling Patient Data in Form

Objective

Ensure MedicalRecord form is pre-filled with correct patient and appointment details.

Requirements

- Fetch patient details using the **appointment ID**.
- Display patient information in a **read-only** format.

Technical

- Use **data binding** for automatic updates.
- Implement a **cache mechanism** for reducing database calls.

Final

- Doctors should see the correct patient details pre-filled in the form.

Task 2.i: Testing MedicalRecord Creation Workflow

Objective

Verify that the entire MedicalRecord creation process works as expected.

Requirements

- Test **UI interactions** (form filling, button clicks).
- Test **data persistence** in the database.
- Simulate various scenarios (valid input, missing fields, errors).

Technical

- Use **unit testing** for backend functions.
- Perform **manual UI testing**.

Final

- MedicalRecord creation should work smoothly with no critical bugs.

Task 2.j: Validating MedicalRecord Database Entries

Objective

Ensure data integrity and correctness of stored MedicalRecords.

Requirements

- Check that all **fields are correctly saved**.
- Verify that each MedicalRecord is linked to the right appointment.
- Ensure no **duplicate entries** exist.

Technical

- Use **SQL queries** to inspect stored data.
- Implement **data validation checks**.

Final

- Data should be accurate and well-structured.

Task 2.k: Error Handling for MedicalRecord Process

Objective

Ensure proper error handling during MedicalRecord creation.

Requirements

- Implement clear **error messages** for incorrect inputs.
- Handle **database failures gracefully**.
- Prevent invalid form submissions.

Technical

- Use **try-catch blocks** in service and repository layers.
- Implement **logging mechanisms** for error tracking.

Final

- The system should be **user-friendly** and **fail-safe**.